XGBoost 论文阅读总结

1. 介绍

XGBoost的可扩展性(scalability)归因于一些重要的系统优化和算法优化。这些优化包括：

- 一种新的tree-learning算法(a novel tree learning algorithm)：用于处理稀疏数据(sparse data)
- 一种理论正确的加权分位数略图过程（a theoretically justified weighted quantile sketch procedure）：用于处理在近似的tree-learning中实例权重

由于XGBoost的**并行化和分布式计算**，使得learning过程比其它模型实现要快。更重要地，XGBoost实现了**核外计算(out-of-core computation: 基于外存)**，使得数据科学家们可以在pc机上处理上亿的训练实例。最终，会把这些技术结合起来实现一个end-to-end的系统，可以扩展到集群上。

主要内容：

- 1.设计和建立了一个高度可扩展的**end-to-end tree boosting系统**
- 2.提出了一种**理论正确的加权分位数略图过程(theoretically justified weighted quantile sketch procedure)**，用于高效地进行预计算
- 3.介绍了一种新的**稀疏感知算法（sparsity-aware algorithm）**，用于并行化tree learning
- 4.提出了一种高效的**内存感知块结构（cache-aware block structure）**，用于核外（out-of-core）tree learning

2. XGBoost 算法

XGBoost的方法源自于Friedman的二阶方法。XGBoost在正则化目标函数上做了最小的改进。

2.1 正则化目标函数

对于一个含**n个训练样本，m个features**的结定数据集：$D = (x_i, y_i)(|D| = n, x_i \in R^m, y_i \in R)$，所使用的tree ensemble model使用**K次求和函数**来预测输出：

$$\hat{y_i} = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), f_k \in F \tag{1}$$

其中，$F = f(x) = w_{q(x)}$，**满足**$(q : R^m \rightarrow T, w \in R^T)$，是回归树(CART)的空间。**q表示每棵树的结构，它会将一个训练样本实例映射到相对应的叶子索引上。T是树中的叶子数。每个f_k对应于一个独立的树结构q和叶子权重w。**与决策树不同的是，每棵回归树包含了在每个叶子上的一个连续分值，**我们使用w_i来表示第i个叶子上的分值**。对于一个给定样本实例，我们会使用树上的决策规则(由q给定)来将它分类到叶子上，并通过将相应叶子上的分值(由w给定)做求和，计算最终的预测值。为了在该模型中学到这些函数集合，我们会对下面的正则化目标函数做最小化：

$$L(\phi) = \sum_i l(\hat{y_i}, y_i) + \sum_i \Omega(f_k) \tag{2}$$

其中：$\Omega(f) = \gamma T + \frac{1}{2}\lambda||\omega||^2$

其中，l 是一个可微凸loss函数（differentiable convex loss function），可以计算预测值 $\hat{y_i}$ 与目标值 y_i 间的微分。第二项 Ω 会惩罚模型的复杂度。正则项可以对最终学到的权重进行平滑，避免overfitting。相类似的正则化技术也用在RGF模型(正则贪婪树)上。XGBoost的目标函数与相应的学习算法比RGF简单，更容易并行化。当正则参数设置为0时，目标函数就相当于传统的gradient tree boosting方法。

2.2 Gradient Tree Boosting

等式（2）中的tree ensemble模型将函数作为参数，不能使用在欧拉空间中的传统优化方法进行优化。模型以一种叠加的方式进行训练。正式地，$\hat{y_i}^{(t)}$ **为第i个实例在第t次迭代时的预测**，我们需要添加 f_t，然后最小化下面的目标函数：

$$L^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

这意味着，我们贪婪地添加 f_t，根据等式(2)尽可能地提升模型。使用二阶近似可以快速优化目标函数。

$$L^{(t)} \backsimeq \sum_{i=1}^{n} [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

其中，$g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$，$h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$分别是loss function上的一阶梯度和二阶梯度。我们可以移除常数项，从而获得如下所示的**在t次迭代时的简化版目标函数：**

$$L^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{3}$$

我们定义$I_j = \{i|q(x_i) = j\}$是叶子 j 的实例集合。将(3)式进行重写，并展开 Ω 项：

$$\begin{aligned} L^{(t)} &= \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned} \tag{4}$$

对于一个确定的结构q(x)，我们可以计算最优的权重 w_j^*：

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{5}$$

代入(5)计算得到对应的loss最优解为：

$$L^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \tag{6}$$

等式(6)可以作为一个得分函数（scoring function）来衡量一棵树结构q的质量（quality）。该分值类似于决策树里的不纯度(impurity score)，只不过它从一个更宽范围的目标函数求导得到。图2展示了该分值是如何被计算的。

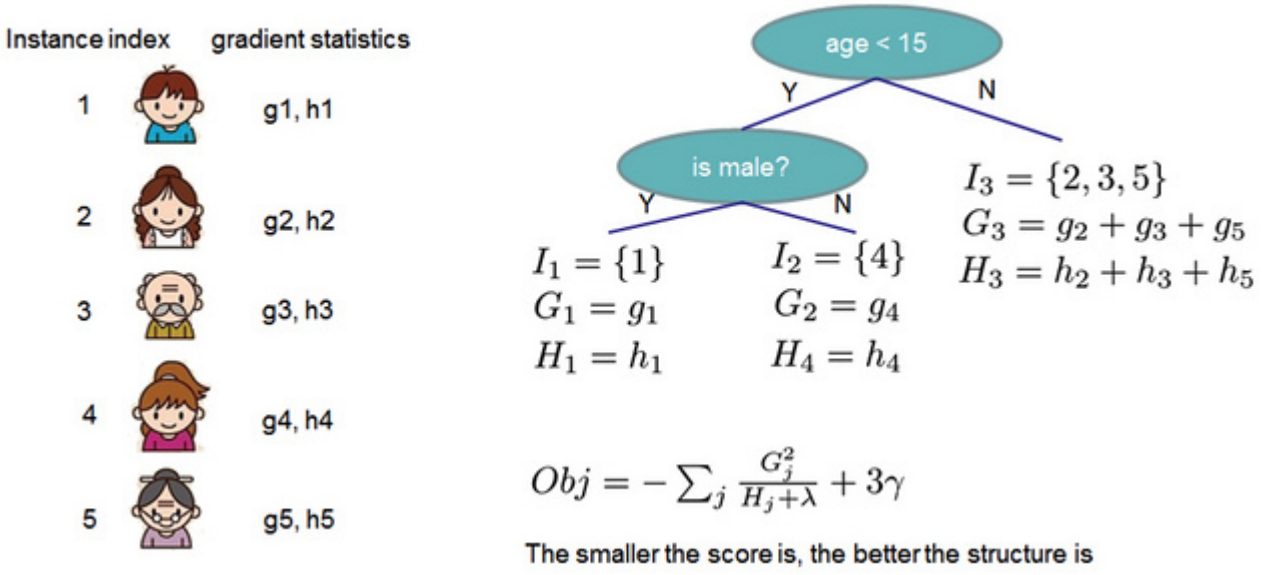


图2:结构得分(structure score)计算。我们只需要在每个叶子上对梯度和二阶梯度统计求和，然后应用得分公式（scoring formula）来获得质量分（quality score）。

通常，不可能枚举所有可能的树结构**q**。而贪婪算法会从单个叶子出发，迭代添加分枝到树中。假设I_L和I_R是一次划分**(split)**后的左节点和右节点所对应的实例集合。$I = I_L \bigcup I_R$，接着，在split之后的loss reduction为：

$$L_{split} = \frac{1}{2}[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda}] - \gamma \tag{7}$$

该式通常在实际中用于评估split的候选（split candidates）。

2.3 Shrinkage和列子抽样(column subsampling)

除了2.1节所提到的正则化目标函数外，还会使用两种额外的技术来进一步阻止overfitting。第一种技术是**Friedman**介绍的**Shrinkage**。Shrinkage会在每一步tree boosting时，会将新加入的**weights**通过一个因子η进行缩放。与随机优化中的learning rate相类似，对于用于提升模型的新增树(future trees)，shrinkage可以减少每棵单独的树、以及叶子空间（leaves space）的影响。第二个技术是列特征子抽样**(column feature subsampling)**。该技术也会在RandomForest中使用，在商业软件TreeNet中的gradient boosting也有实现，但开源包中没实现。根据用户的反馈，比起传统的行子抽样（row sub-sampling：同样也支持），使用列子抽样可以阻止overfitting。列子抽样的使用可以加速并行算法的计算(后面会描述)。

3. SPLIT FINDING ALGORITHMS

3.1 Basic Exact Greedy Algorithm

tree learning的其中一个关键问题是，找到等式(7)的最好划分(best split)。为了达到这个目标，**split finding算法**会在所有特征（features）上，枚举所有可能的划分（splits）。我们称它为**“完全贪婪算法(exact greedy algorithm)”**。许多单机版tree-boosting实现中，包括scikit-learn，R's gbm以及单机版的XGBoost，都支持完全贪婪算法(exact greedy algorithm)。该算法如算法1所示。它会对连续型特征（continuous features）枚举所有可能的split。为了更高效，该算法必须首先根据特征值对数据进行排序，以有序的方式访问数据来枚举等式(7)中的结构得分（structure score）的梯度统计(gradient statistics)。

Algorithm 1: Exact Greedy Algorithm for Split Finding

Input: I, instance set of current node
Input: d, feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ **to** m **do**
  $G_L \leftarrow 0$, $H_L \leftarrow 0$
  for j *in sorted*(I, *by* $\mathbf{x}_{jk}$) **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split with max score

https://blog.csdn.net/Dby_freedom

[算法1]

(1) for j in sorted(I, by x_{jk})：表示对第m个特征，对叶子节点j，依据特征k进行对该节点内的所有实例进行从小到大的排序，依据下面Score公式计算该split下的Score。

(2) 算法总结：对每个叶子节点j，对叶子节点的每个特征k，依据k对实例进行排序，然后计算该特征下的Score，对该叶子节点，遍历所有特征后，得到该节点下对应的score及split进行输出（即最佳割特征下对应的最佳分割点）。

(3) However, it is impossible to efficiently do so when the data does not fit entirely into memory.

3.2 Approximate Algorithm

前面提到过，XGBoost每一步选能使分裂后增益最大的分裂点进行分裂。而分裂点的选取之前是枚举所有分割点，这称为精确的贪心法（Exact Greedy Algorithm）.

当数据量十分庞大，以致于不能全部放入内存时，Exact Greedy 算法就会很慢。 因此XGBoost引入了近似的算法。

我们总结了一个近似框架（approximate framework），重组了在文献[17,2,22]中提出的思想，如算法2所示。为了进行总结(summarize)，**该算法会首先根据特征分布的百分位数(percentiles of feature distribution)，提出候选划分点(candidate splitting points)。接着，该算法将连续型特征映射到由这些候选点划分的分桶(buckets)中，聚合统计信息，基于该聚合统计找到在建议（proposal）间的最优解。**

简单的说，就是根据特征 k 的分布来确定 l 个候选切分点 $S_k = \{s_{k1}, s_{k2}, \ldots, s_{kl}\}$，然后根据这些候选切分点把相应的样本放入对应的桶中，对每个桶的G,H进行累加。最后在候选切分点集合上贪心查找，和Exact Greedy Algorithm类似。该算法描述如下：

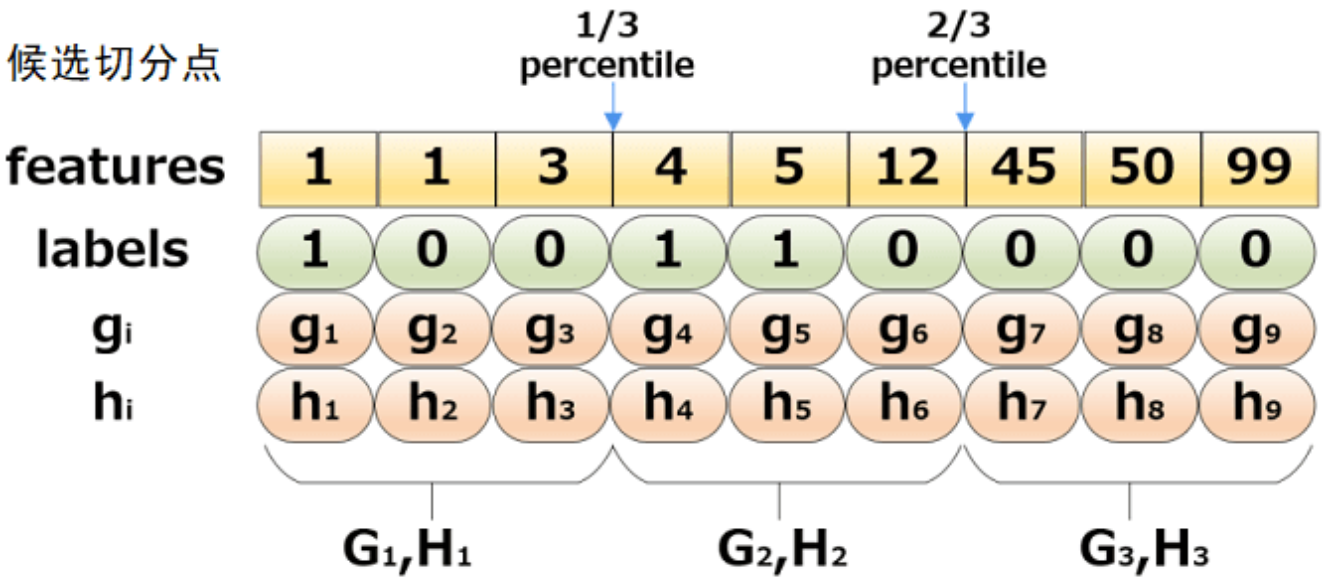
Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ *to* m **do**
  Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on featu
  Proposal can be done per tree (global), or per split(loc
end
for $k = 1$ *to* m **do**
  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} h_j$
end
Follow same step as in previous section to find max score only among proposed splits.

[算法2]

> 算法讲解： 第一个`for`循环做的工作：对特征`k`根据该特征分布的分位数找到切割点的候选集合 $S_k = s_{k1}, s_{k2}, \ldots, s_{kl}$ ，这样做的目的是提取出部分的切分点不用遍历所有的切分点。其中获取某个特征`k`的候选切割点的方式叫`proposal`。主要有两种`proposal`方式：`global proposal`和`local proposal`。 第二个`for`循环的工作：将每个特征的取值映射到由这些该特征对应的候选点集划分的分桶(buckets)区间 $\{s_{k,v} \geq x_{jk} > s_{k,v-1}\}$中，对每个桶（区间）内的样本统计值 G, H 进行累加统计，最后在这些累计的统计量上寻找最佳分裂点。这样做的主要目的是获取每个特征的候选分割点的 G, H 量。

给定了候选切分点后，一个例子为：



$$Gain = \max\{Gain, \frac{G_1^2}{H_1+\lambda} + \frac{G_{23}^2}{H_{23}+\lambda} - \frac{G_{123}^2}{H_{123}+\lambda} - \gamma, \frac{G_{12}^2}{H_{12}+\lambda} + \frac{G_3^2}{H_3+\lambda} - \frac{G_{123}^2}{H_{123}+\lambda} - \gamma\}$$

https://blog.csdn.net/Dby_freedom

分桶当然是越多越好，但越多意味计算量也就更大，极端情况下就是一个数据一个桶，但这违背了分桶的初衷。所有，需要综合balance考虑。

那么，现在有两个问题：

1. 如何选取候选切分点 $S_k = \{s_{k1}, s_{k2}, \ldots, s_{kl}\}$ 呢？
2. 什么时候进行候选切分点的选取？

第1个问题在下一小节说明，先回答第2个问题。

还有一种近似的方法，就是不逐一过样本，而是对于当前特征，选择几个分位点，利用这些分位点来将连续特征值映射成独立的分桶，然后用这些聚合后的信息来进行后续的分裂。

分界点选取时机

对于问题2，XGBoost具体实施时，有两者方式：global variant 和local variant ，前者是在每棵树开始之前就分好，后续不再改变；而后者是在每个节点上都具体情况具体分析的考虑。

- Global： **学习每棵树前**， 提出候选切分点
- Local： **每次分裂前**， 重新提出候选切分点

由于global variant后续不再改变，因此往往需要更多的候选点。（However, usually more candidate points are needed for the global proposal because candidates are not refined after each split.）

当然，后者更好，但带来的复杂度也更大。通常情况下，都会优先选用global的形式。

可以预见到的是，local不用像global方式那样设定分桶分的很细，因为它是有针对性的设置，可以在很粗粒度上就达到相同的效果。

对比如下：

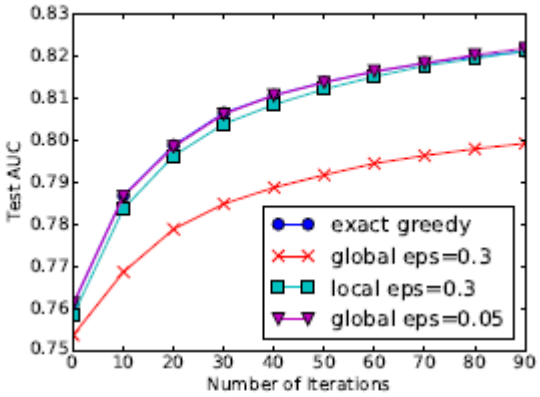


Figure 3: Comparison of test AUC convergence on Higgs 10M dataset. The eps parameter corresponds to the accuracy of the approximate sketch. This roughly translates to 1 / eps buckets in the proposal. We find that local proposals require fewer buckets, because it refine split candidates.

https://blog.csdn.net/Dby_freedom

从实验结果可以看到：

（1） 在全局分桶分的很细时，和exact greedy效果几乎完全相同。（紫色，蓝色）

（2） 当local较大时，几乎可以达到global较小值一样的效果。（紫色，绿色）

（3） 当global设置过大时，整体效果就很差了。（蓝色，红色）

此外，桶的个数等于 1 / eps， 不难得出结论:

- 全局切分点的个数够多的时候，和Exact greedy算法性能相当。
- 局部切分点个数不需要那么多，因为每一次分裂都重新进行了选择。

3.3 Weighted Quantile Sketch(加权分位数略图)

对于问题1，可以采用分位数，也可以直接构造梯度统计的近似直方图等。

先补充一下分位数相关知识：

3.3.1 分位点及分位数补充介绍

分位点 (quantile) 介绍：

举例子说明，一般来说对于一个数据列表有：

> input：14, 19, 3, 15, 4, 6, 1, 13, 13, 7, 11, 8, 4, 5, 15, 2 sort： 1, 2, 3, 4, 4, 5, 6, 7, 8, 11, 13, 13, 14, 15, 15, 19
> rank： 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

`φ-quantile`表示 `rank= φ-quantile × N`的元素。我们需要取出`0.25-quantile`（0.25分位点的数），即取出 rank 为 $0.25 \times N$ 的数。

加权分位点 (Weighted quantile) 介绍：

举例子说明，一般来说对于一个数据列表有：

> input： 1, 2, **3**, 4, 5 weight：0.1, 0.2, 0.2, 0.3, 0.5 rank： 0.1, 0.3, **0.5**, 0.8, 1.3

取元素的方法为：指定 rank（大于0 小于1）即可，如 0.5 分位点的元素为3。

分位数缩略图 (Quantile Sketch)

原因： 当一个序列无法全部加载到内存时，常常采用分位数缩略图近似的计算分位点，以近似获取特定的查询。

方式： 使用随机映射 (Random projections) 将数据流投射在一个小的存储空间内作为整个数据流的概要，这个小空间存储的概要数据称为略图，可用于近似回答特定的查询。**需要保留原序列中的最小值和最大值。**

近似算法中的分位数缩略图 (Weighted Quantile Sketch)

如何根据特征分布的分位数挑选出候选点集？

通常一个特征的百分位数可以被用来让候选在数据上进行均匀地分布。**创建**一个 **multi-set**：
$D_k = \{(x_{1k}, h_1), (x_{2k}, h_2), \ldots, (x_{nk}, h_n)\}$， n 是样本个数。

> D_k表示第 k 个特征$feature_k$ 与二阶偏导 H 之间的集合【每个训练实例的第 k 个特征值以及它的二阶梯度值统计。其中 h_i 表示 i 个实例的第 k 个特征值对应的二阶梯度值统计，可看作 i 个实例的第 k 个特征值的权重。】

一般来说对于加权分位数缩略图的取值是根据 Rank 值进行的，Rank 计算公式如下：

定义序列函数 r_k：$r_k(z) = \frac{\sum_{(x,h)\in D_k, x<z} h}{\sum_{(x,h)\in D_k} h}$

> 该`Rank`函数，输入为某个特征值`z`，计算的是该特征所有可取值中小于`z`的特征值的总权重 占总的所有可取值的总权重和的比例，输出为一个比例值。

于是就使用下面这个不等式**寻找候选分割点** $\{s_{k1}, s_{k2}, \ldots, s_{kl}\}$

$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \epsilon$， $s_{k1} = \min_i x_{ik}, s_{kl} = max_i x_{ik}$

> 其中： s_{k1} 是特征 k 的取值中最小的值 x_{ik}, s_{kl} 是特征 k 的取值中最大的值 x_{ik}，这是分位数缩略图的要求需要保留原序列中的最小值和最大值。ϵ 是一个近似比例，或者说是扫描步幅。可以理解为在特征 k 的取值范围上，按照步幅，挑选出特征 ϵ 的取值候选点，组成候选点集。起初是从 s_{k1} 起，每次增加 $\epsilon \times (s_{kl} - s_{k1})$ 作为候选点，加入到候选集中。如此计算的话，这意味着大约是 $1/\epsilon$个候选点。

此时特征 k 的取值中 $\min_i x_{ik}, \max_i x_{ik}$ 来自 **multi-set** D_k，对于 D_k 的数据集有两种定义：

> (1) 一开始选好，然后每次树切分都不变，也就是说是在总体样本里选 $\min_i x_{ik}, \max_i x_{ik}$，这就是我们之前定义的`global proposal`； (2) 是树每次确定好切分点的分割后样本也需要进行分割，$\min_i x_{ik}, \max_i x_{ik}$来自子树的样本集 D_k ，这就是`local proposal`。

关于`global & local`来自 解析XGBoost 的解读：

> 全局变种（**global variant**） 会在树构建的初始阶段，建议所有的候选划分，并在所有的层级（level）上使用相同的建议。局部变种（**local variant**）则在每次划分后重新建议（re-proposes）。比起局部法，全局法需要更少的建议步骤。然而，对于全局建议，通常需要更多的候选点，因为在每次划分之后，不需要重新定义候选。局部建议会在每次划分后重新定义候选，对于更深的树更合适。原文图3展示了在Higgs boson数据集上不同算法的比较。发现局部建议确实需要更少的候选。如果两者的候选一样多，全局建议比局部建议会更精确。

> 注：global variant 与 local variant 的差别主要体现在候选点建议上，global variant 想要一劳永逸，在树构建的初始阶段，直接提出所有的候选可能切分点，以后进行进行split finding 就直接在这些候选点中进行增益计算，进而得到每个节点下的最佳切分特征及最佳切分点；而local variant 则是采用每次进行节点split finding 时，重新确认候选可能切分点，重新确认候选的过程与global variant 一样，但由于各个节点覆盖的样本不一样了，故而其重新确定候选可以带来更精确的候选划分，带也大大提升了计算量。

而XGBoost不单单是采用简单的分位数的方法，而是对分位数进行加权（使用二阶梯度**h**），称为：Weighted Quantile Sketch。PS:上面的那个例子采用的是没有使用二阶导加权的分位数。

对特征 k 构造multi-set 的数据集：$D_k = (x_{1k}, h_1), (x_{2k}, h_2), \ldots, (x_{nk}, h_n)$, 其中 x_{ik}表示样本 i 的特征 k 的取值，而 h_i 则为对应的二阶梯度。

可以定义一个rank function为：

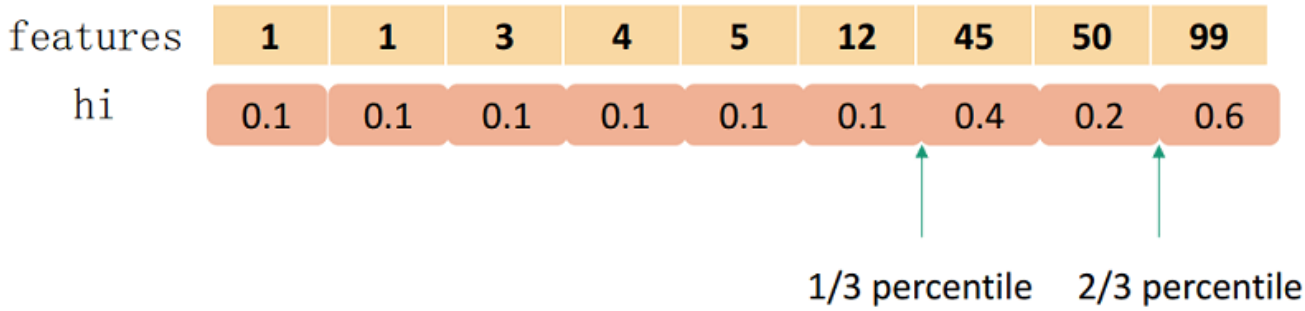
$$r_k(z) = \frac{1}{\sum_{(x,h)\in D_k} h} \sum_{(x,k)\in D_k, x<z} h$$

它表示相应第**k**个特征上的输入值小于**z**的实例的占比。和之前的分位数挺相似，不过这里是按照二阶梯度进行累计。而候选切分点 $\{s_{k1}, s_{k2}, \ldots, s_{kl}\}$要求：

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \varepsilon, \qquad s_{k1} = \min_i x_{ik},\ s_{kl} = \max_i x_{ik}$$

用大白话说就是让相邻两个候选分裂点相差不超过某个值 ϵ。因此，总共会得到 $\frac{1}{\epsilon}$ 个切分点。

一个例子如下：



其中 ϵ 是近似因子（approximation factor）。直觉上，这意味着大约是 $\frac{1}{\epsilon}$ 个候选点。这里，**每个数据点通过 h_i 加权**。为什么h_i可以表示权重呢？我们可以重写(3)式：

要切分为3个，总和为1.8，因此第1个在0.6处，第2个在1.2处。

那么，**为什么要用二阶梯度加权**？将前面我们泰勒二阶展开后的目标函数2-4进行配方：

$$
\begin{aligned}
&\sum_{i=1}^{N}\left(g_i f_t(\mathbf{x_i})+\frac{1}{2}h_i f_t^2(\mathbf{x_i})\right)+\Omega(f_t) \qquad (8)\\
&=\sum_{i=1}^{N}\frac{1}{2}h_i\left(2\frac{g_i}{h_i}f_t(\mathbf{x_i})+f_t^2(\mathbf{x_i})\right)+\Omega(f_t)\\
&=\sum_{i=1}^{N}\frac{1}{2}h_i\left(\frac{g_i^2}{h_i^2}+2\frac{g_i}{h_i}f_t(\mathbf{x_i})+f_t^2(\mathbf{x_i})\right)+\Omega(f_t)\\
&=\sum_{i=1}^{N}\frac{1}{2}h_i\left(f_t(\mathbf{x_i})-(-\frac{g_i}{h_i})\right)^2+\Omega(f_t)
\end{aligned}
$$

推导第三行可以加入 $\frac{g_i^2}{h_i^2}$ 是因为 g_i 和 h_i 是上一轮的损失函数求导，是常量。

从式8可以看出，**就像是标签为 $-g_i/h_i$，权重为 h_i 的平方损失，因此用 h_i 加权**。

PS: 原论文的g_i/h_i符号错了，我推导的时候觉得很奇怪，查了很多介绍XGBoost的资料，都没有说明如何推导，直接把原公式一贴，这是很不好的。最后看到了Stack Exchange上的回答：

> The second equation should have its **sign reversed**, as in:
>
> $$
> \begin{aligned}
> &\sum_{i=1}^{N}\frac{1}{2}h_i[f_t(x_i)-(-g_i/h_i)]^2+constant\\
> &=\sum_{i=1}^{N}\frac{1}{2}h_i[f_t^2(x_i)+2\frac{f_t(x_i)g_i}{h_i}+(g_i/h_i)^2]\\
> &=\sum_{i=1}^{N}[g_i f_t(x_i)+\frac{1}{2}h_i f_t^2(x_i)+\frac{gi^2}{2h_i}]
> \end{aligned}
> $$
>
> The last term is indeed constant: remember that the g_i and h_i are determined by the previous iteration, so they're constant when trying to set f_t.
>
> So, now we can claim "this is exactly weighted squared loss with labels $-g_i/h_i$ and weights h_i

> Credit goes to Yaron and Avi from my team for explaining me this.
>
> ——from Need help understanding xgboost's approximate split points proposal

为了解决该问题，XGBoost引入了一种新的分布式加权分位数略图算法（distributed weighted quantile sketch algorithm），使用一种可推导证明的有理论保证的方式，来处理加权数据。**总的思想是，提出了一个数据结构，它支持merge和prune操作，每个操作证明是可维持在一个固定的准确度级别**。算法的详细描述在 这里。

3.4 Sparsity-aware Split Finding

在许多现实问题中，输入x是稀疏的。有多种可能的情况造成稀疏：

- 数据中的missing values
- 统计中常见的0条目
- 特征工程：比如one-hot encoding

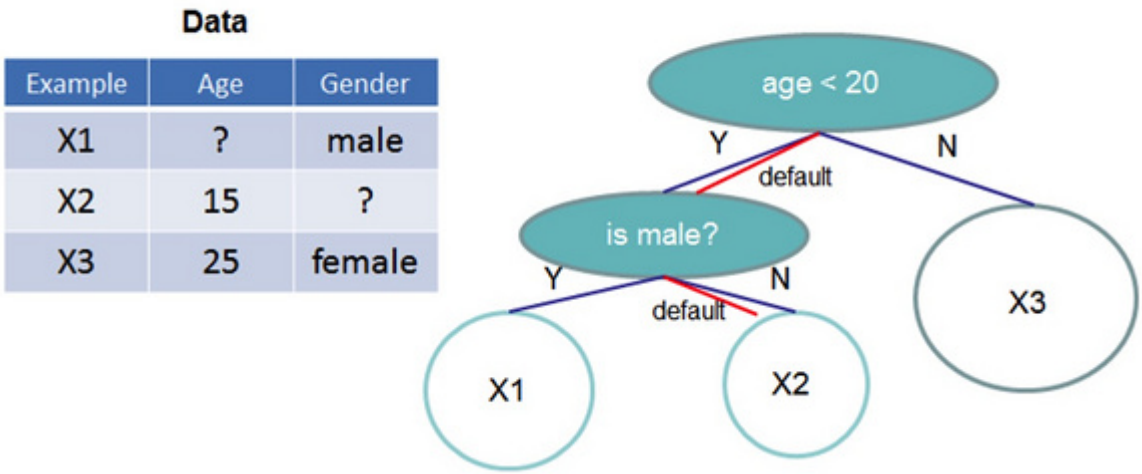


图4: 带缺省方向的树结构。当在split时相应的feature缺失时，一个样本可以被归类到缺省方向上

让算法意识到数据中的稀疏模式很重要。为了这么做，我们提出了在每个树节点上增加一个 **缺省的方向（default direction）**，如图4所示。当稀疏矩阵x中的值缺失时，样本实例被归类到缺省方向上。在每个分枝上，缺省方向有两种选择，**最优的缺省方向可以从数据中学到。**

如算法3所示。关键的改进点是：**只访问非缺失的条目** I_k 。上述算法会将未出现值（non-presence）当成是一个missing value，学到最好的方向来处理 missing values。

XGBoost能对缺失值自动进行处理，其思想是**对于缺失值自动学习出它该被划分的方向**（左子树or右子树）

Algorithm 3: Sparsity-aware Split Finding

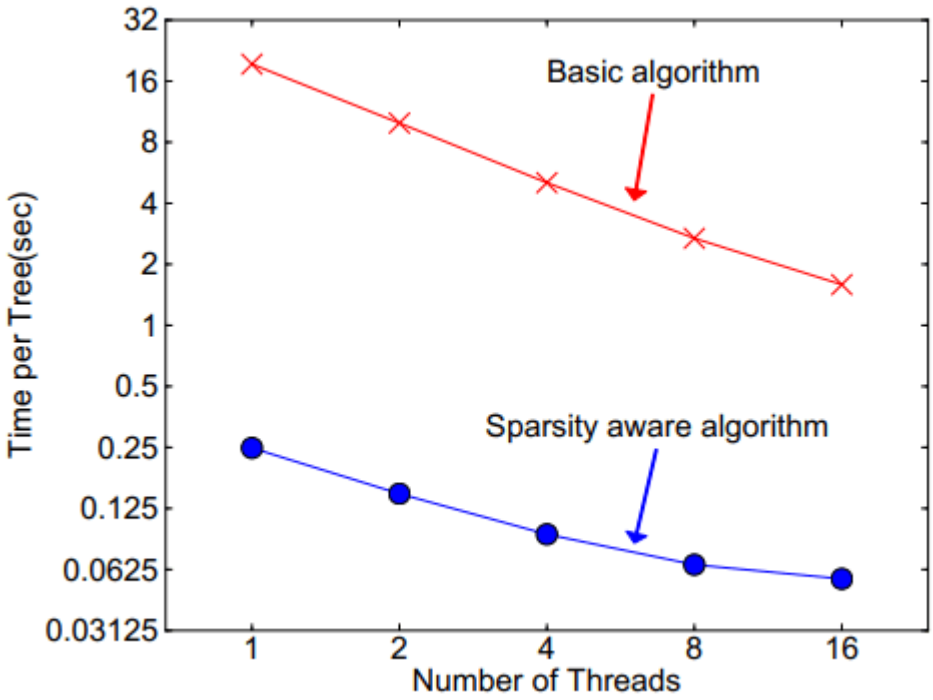
Input: I, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d, feature dimension
Also applies to the approximate setting, only collect statistics of non-missing entries into buckets
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  // enumerate missing value goto right
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted*$(I_k,$ *ascent order by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
  // enumerate missing value goto left
  $G_R \leftarrow 0,\ H_R \leftarrow 0$
  for j *in sorted*$(I_k,$ *descent order by* $\mathbf{x}_{jk})$ **do**
    $G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$
    $G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split and default directions with max gain

https://blog.csdn.net/Thy_freedom

注意，**上述的算法只遍历非缺失值**。划分的方向怎么学呢？很naive但是很有效的方法：

1. 让特征k的所有缺失值的都到右子树，然后和之前的一样，枚举划分点，计算最大的gain
2. 让特征k的所有缺失值的都到左子树，然后和之前的一样，枚举划分点，计算最大的gain

这样最后求出最大增益的同时，也知道了缺失值的样本应该往左边还是往右边。使用了该方法，相当于比传统方法多遍历了一次，但是它只在非缺失值的样本上进行迭代，因此其复杂度与非缺失值的样本成线性关系。在Allstate-10k数据集上，比传统方法快了50倍:

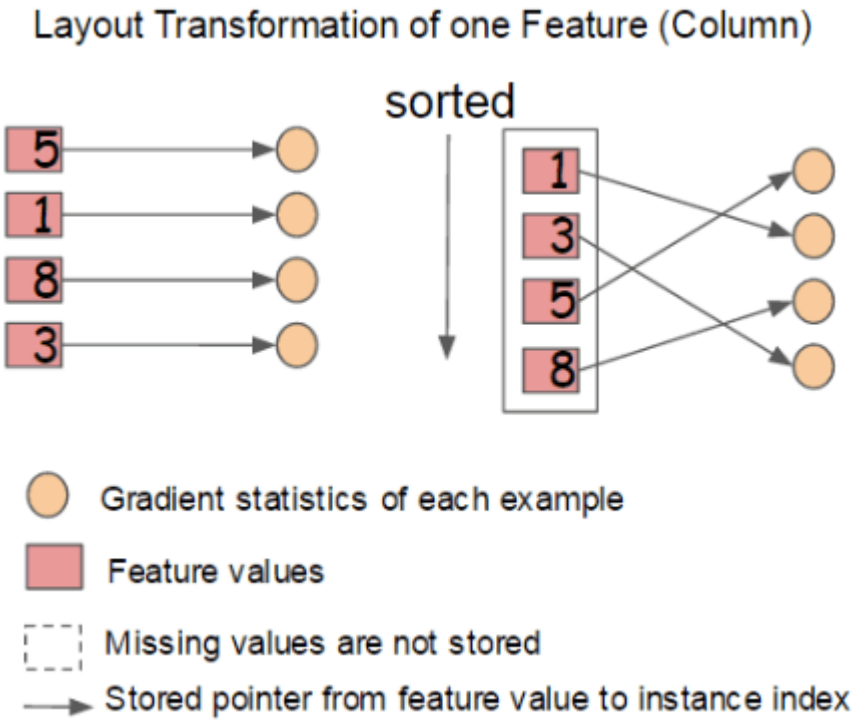


4. SYSTEM DESIGN

4.1 Column Block for Parallel Learning(用于并行学习的Column Block)

在建树的过程中，最耗时是找最优的切分点，而这个过程中，最耗时的部分是将数据排序。为了减少排序的时间，提出Block结构存储数据。

- Block中的数据以稀疏格式**CSC**进行存储
- Block中的**特征进行排序**（不对缺失值排序）
- Block 中特征还需存储指向样本的**索引**，这样才能根据特征的值来取梯度。
- 一个Block中存储一个或多个特征的值



可以看出，**只需在建树前排序一次**，后面节点分裂时可以直接根据索引得到梯度信息。

时间复杂度分析

d为树的最大深度，K为树的总树目。对于exact greedy algorithm，原始的稀疏感知算法的时间复杂度：

$$O(Kd|x|_0logn)$$

> 这里，我们使用 $|x|_0$ 来表示在训练数据中未缺失条目（non-missing entries）的数目。其中$|x|_0logn$是特征排序时间复杂度，而 Kd 则是 $数的总数目 * 树的深度 = counts\ of\ split\ finding$，当不做任何优化时，对每棵树，每层，每个节点进行split finding时，对每个节点的所有特征对应的样本进行该特征下的大小排序，故而总的时间复杂度为上述步骤连乘。
>
> 而 $Kd|x|_0$ 则是进行split finding 的时间复杂度开销，最多进行 $Kd|x|_0$ 次查找（对应每个样本为一个节点）。

另一方面，块结构上的tree boosting的开销为：

$$O(Kd|x|_0 + |x|_0logn)$$

这里， $O(\|x\|_0logn)$ 是一次预处理开销(one time preprocessing cost)(即排序开销)，可以分期(be amortized)。

> 因为采用了block structure之后，对于exact greedy algorithm 就是采用的是先对所有特征进行列排序，然后每次进行split finding时候，直接在block spliting 中进行查找线型遍历，得到各节点所有特征所有划分的对应增益，选择最大增益对应的特征及划分即为所求。
>
> 而 $Kd|x|_0$ 则是进行split finding 的时间复杂度开销，最多进行 $Kd|x|_0$ 次查找（对应每个样本为一个节点）。

该分析展示了块结构可以帮助节省一个额外的$logn$因子，其中当n非常大时就很大。

对于近似算法，使用二分查找的原始算法时间复杂度为：

$$O(Kd|x|_0logq)$$

这里的q是在数据集中建议候选的数目。其中，q通常为32~100之间，log因子仍会引入间接开销。使用块结构，我们可以将时间减小到：

$$O(Kd|x|_0 + |x|_0 logB)$$

其中B是在每个块中的行的最大数。同样的，我们可以在计算中节约额外的log q因子。

> 解释同上，只是采用了近似算法后，采用了本论文提出的 Weighted Quantile Sketch 进行样本点候选筛选，这样即将原始样本点 n 降到候选点 q。

- 在Exact greedy算法中，将整个数据集存放在一个Block中。这样，复杂度从原来的$O(Hd||x||_0 \log n)$ 降为$O(Hd||x||_0 + ||x||_0 \log n)$，其中$||x||_0$为训练集中非缺失值的个数。这样，Exact greedy算法就省去了每一步中的排序开销。
- 在近似算法中，使用多个Block，每个Block对应原来数据的子集。不同的Block可以在不同的机器上计算。该方法对Local策略尤其有效，因为Local策略每次分支都重新生成候选切分点。

Block结构还有其它好处，数据按列存储，可以同时访问所有的列，很容易实现并行的寻找分裂点算法。此外也可以方便实现之后要讲的out-of score计算。

缺点是空间消耗大了一倍。

> 利用列块进行并行计算：在我们训练过程中我们主要是做分支处理，分支处理就要对每一列（特征）找出适合的分裂点。通常来说，我们更青睐使用csc存储，这样我们就方便取出来。再者我们在分支的时候都会预先对数据按照其特征值进行排序。所以我们将数据按照列存储成一个数据块方便我们在分支的时候并行处理。所以我们要知道**XGB**的并行计算的粒度不在树上，而是在特征上，尤其是不同分支节点上（**leaf-wise**）。当然这也成为XGB的一个问题所在，需要额外的空间存储pre-sort的数据。而且每次分支后，我们都要找处落在下一个子节点上的样本，并组织好它。后来就有了**LightGBM**，下次我再将其整理出来。

4.2 Cache-aware Access

使用Block结构的一个缺点是取梯度的时候，是通过索引来获取的，而这些梯度的获取顺序是按照特征的大小顺序的。这将导致非连续的内存访问，可能使得CPU cache缓存命中率低，从而影响算法效率。

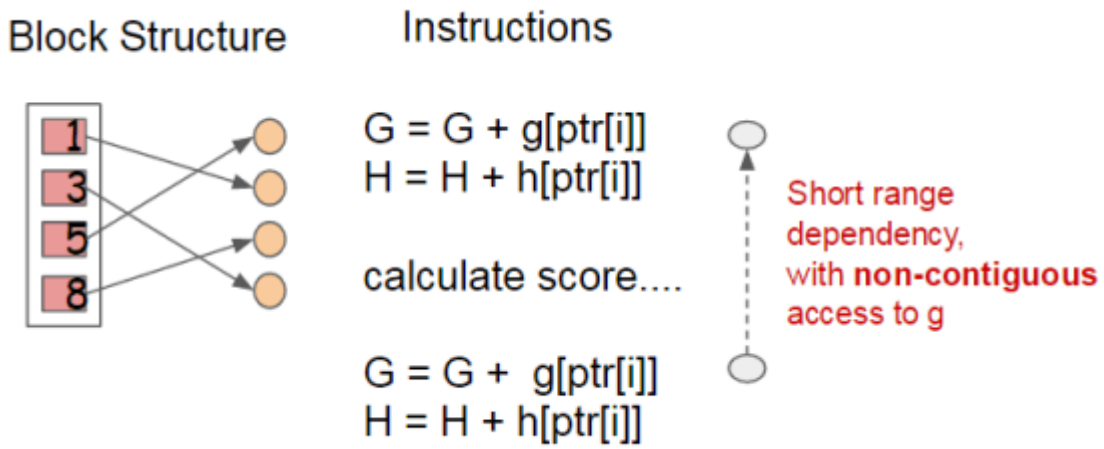
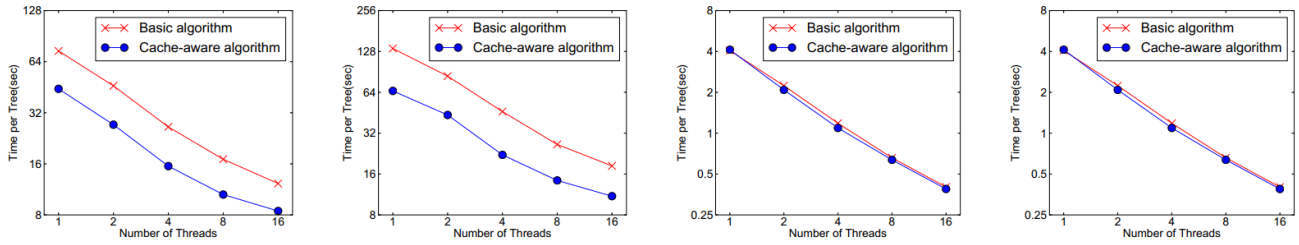


图8: 短范围内的数据依赖模式，由于cache miss，可引起停转（stall）

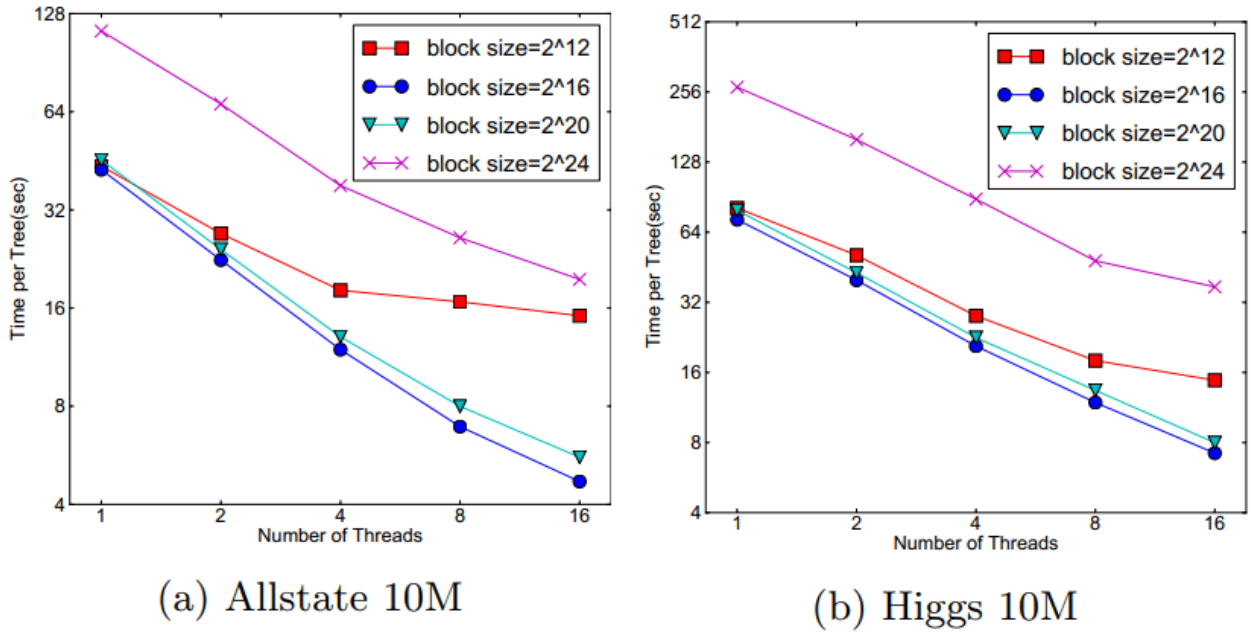
因此，对于exact greedy算法中, 使用缓存预取。具体来说，对每个线程分配一个连续的buffer，读取梯度信息并存入Buffer中（这样就实现了非连续到连续的转化），然后再统计梯度信息。该方式在训练样本数大的时候特别有用，见下图：



(a) Allstate 10M (b) Higgs 10M (c) Allstate 1M (d) Higgs 1M

图7: 在exact greedy algorithm中，cache-aware prefetching的影响。我们发现，cache-miss会在大数据集（1000w实例）上影响性能。使用cache-aware prefetching，可以提升数据集很大时的性能。

在approximate 算法中，对Block的大小进行了合理的设置。定义Block的大小为Block中最多的样本数。选择一个过小的block size会导致每个thread会小负载（small workload）运行，并引起低效的并行化(inefficient parallelization)。在另一方面，过大的block size会导致cache miss，梯度统计将不能装载到CPU cache中。block size的好的选择会平衡两者。设置合适的大小是很重要的，设置过大则容易导致命中率低，过小则容易导致并行化效率不高。经过实验，发现2^16比较好。



(a) Allstate 10M (b) Higgs 10M

图9: 在近似算法中，block size的影响。我们发现，过小的块会引起并行化很低效，过大的块由于cache miss会让训练慢下来

> 缓存处理能力：对于有大量数据或者说分布式系统来说，我们不可能将所有的数据都放进内存里面。因此我们都需要将其放在外存上或者分布式存储。但是这有一个问题，这样做每次都要从外存上读取数据到内存，这将会是十分耗时的操作。因此我们使用预读取（**prefetching**）将下一块将要读取的数据预先放进内存里面。其实就是多开一个线程，该线程与训练的线程独立并负责数据读取。此外，我还要考虑block的大小问题。如果我们设置最大的block来存储所有样本在k特征上的值和梯度的话，cache未必能一次性处理如此多的梯度做统计。如果我们设置过少block size，这样不能充分利用的多线程的优势，也就是训练线程已经训练完数据，但是prefetching thread还没把数据放入内存或者cache中。经过测试，作者发现block size设置为2^16个examples最好：

4.3 Blocks for Out-of-core Computation

当数据量太大不能全部放入主内存的时候，为了使得out-of-core计算称为可能，将数据划分为多个Block并存放在磁盘上。

- 计算的时候，使用独立的线程预先将Block放入主内存，因此可以在计算的同时读取磁盘
- Block压缩，貌似采用的是近些年性能出色的LZ4 压缩算法，按列进行压缩，读取的时候用另外的线程解压。对于行索引，只保存第一个索引值，然后用16位的整数保存与该block第一个索引的差值。
- Block Sharding， 将数据划分到不同硬盘上，提高磁盘吞吐率

块压缩（Block Compression） 块通过列(column)进行压缩，当加载进主存时可以由一个独立的线程即时解压(decompressed on the fly)。它会使用磁盘读开销来获得一些解压时的计算。我们使用一个通用目的的压缩算法来计算特征值。对于行索引（row index），我们从块的起始索引处开始抽取行索引，使用一个16bit的整数来存储每个偏移(offset)。**这需要每个块有216216个训练样本，这证明是一个好的设置**。在我们测试的大多数数据集中，我们达到大约26% ~ 29%的压缩率。

块分片（Block Sharding） 第二个技术是，在多个磁盘上以一种可选的方式共享数据。一个pre-fetcher thread被分配到每个磁盘上，取到数据，并装载进一个in-memory buffer中。训练线程（training thread）接着从每个bufer中选择性读取数据。当提供多个磁盘时，这可以帮助增加磁盘读(disk reading)的吞吐量。

> 数据块以外的计算力提高：对于超大型的数据，我们不可能都放入放入内存，因此大部分都放入外存上。假如我们将数据存于外存上将给我们带来读写速度受限的问题。文中有两种方法，一种是对数据进行压缩存于外存中，到内存中需要训练时再解压，这样来增加系统的吞吐率，尽管消耗了一些时间来做编码和解码但还是值得的。另一种就是多外存存储，其实本质上就是分布式存储。这样说有多个线程对分布式结构管理，吞吐率自然高啦。

5. 总结

下面总结几个问题：

5.1 XGBoost为什么快

- 当数据集大的时候使用近似算法
- Block与并行
- CPU cache 命中优化
- Block预取、Block压缩、Block Sharding等

5.2 XGBoost与传统GBDT的不同

这里主要参考weapon的回答，答案在：机器学习算法中GBDT和XGBOOST的区别有哪些？

1. 传统GBDT以CART作为基分类器，XGBoost还支持线性分类器，这个时候XGBoost相当于带L1和L2正则化项的Logistic回归（分类问题）或者线性回归（回归问题）。

   > 注：Boosting 模型本质还是加性模型，而XGBoost做为boosting算法的一种并行实现，除了对传统的GBDT进行了很多改进之外，又增加了其扩展性，其中一个就是支持不同的基分类器，不再局限于CART回归树；
   >
   > 因为XGBoost可以自己设定损失函数，只需要满足该设定损失函数一阶、二阶可导，而损失函数旨在计算真实值与分类器预测值之间的差异。故而，当采用线型模型（包括线型回归和Logistic Regression）采用线性加和的方式进行预测 $\hat{y}_i = \sum_j w_j x_{ij}$，这里的预测值 y 可以由不同的解释，比如我们可以把它作为回归目标的输出，或者进行 sigmoid 变换得到概率（即用 $\frac{1}{1+e^{-\hat{y}_i}}$ 来预测正例的概率），或者作为排序的指标等。此时，基模型即采用的线型模型。

> 此时，XGBoost的正则项 $\Omega(f_t) = \gamma T + \frac{1}{2}\gamma \sum_{j=1}^{T} w_j^2$，此时 T 用于限制线型分类器个数，而 w_j^2 则是用于限制线型分类器预测分数（相较于传统的L2正则化，多了一项基分类器的个数正则项）

2. 传统的GBDT只用了一阶导数信息（使用牛顿法的除外），而XGBoost对损失函数做了二阶泰勒展开。并且XGBoost支持自定义损失函数，只要损失函数一阶、二阶可导。
3. XGBoost的目标函数多了正则项， 相当于**预剪枝**，使得学习出来的模型更加不容易过拟合。
4. XGBoost还有**列抽样**，进一步防止过拟合。
5. 对缺失值的处理。对于特征的值有缺失的样本，XGBoost可以自动学习出它的分裂方向。
6. XGBoost工具支持并行。当然这个并行是在特征的粒度上，而非tree粒度，因为本质还是boosting算法。

> xgboost工具支持并行。boosting不是一种串行的结构吗?怎么并行的？注意xgboost的并行不是tree粒度的并行，xgboost也是一次迭代完才能进行下一次迭代的（第t次迭代的代价函数里包含了前面t-1次迭代的预测值）。xgboost的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），xgboost在训练之前，预先对数据进行了排序，然后保存为block结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个block结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行。

5.4 XGBoost 与 线性回归的区别

1. 线性回归模型的解释性是决策树、随机森林、xgboost无法比拟的，也无法取代。
2. 线性回归可以建立线性模型，而xgboost是不可以的。举个例子，即使是简单的 $y = x + 1$ 的线性关系，xgboost也无法做到。
3. 线性模型计算简单，适用于快速部署。
4. 决策树/森林回归是把数据空间非线性地分成 N 份（N为叶节点数），相当于做N 聚类。然后再每个叶节点上求平均，其实是一种条件期望（conditional mean），$y = E(y|c)$，c 是聚类标签，可以看做是内插（interpolation）。所以预测点周围需要被训练点包围，这样才会有好的效果。
5. 线性模型回归是用线/平面去拟合所有训练数据。当引入 x^2 等高维变量后，在原有 x 空间是非线性曲线/曲面。线性模型回归可以作内插（interpolation，预测点曾经见过），也可以做外插（extrapolation，预测点从没见过，比如 $y = x + 1$）。
6. 决策树优点：决策树用垂直线段去拟合曲线，不用显示的写出曲线的表达式。而线性模型会尝试写出曲线的表达式。所以线性模型更难去拟合复杂的曲线。
7. 决策树缺点：决策树是基于数据，而线性模型是基于模型。决策树需要大量数据去尽量覆盖所有可能的输入，适合于大数据。所谓的“大”是指训练样本的覆盖面要大。线性模型可以处理训练样本小的情况。
8. 决策树和线性模型的组合。在每个叶节点上，单纯的决策树回归是用一个点代表所有值。此时可以再用线性模型，用一条线去拟合子空间的数据。

5.3 XGBoost Scalable的体现

XGBoost的paper在KKD上发表，名为：《Xgboost: A scalable tree boosting system》，那么scalable体现在哪?

参考知乎上王浩的回答，修改如下：

- 模型的scalability：弱分类器可以支持cart也可以支持lr和linear， 但其实这是Boosting算法做的事情，XGBoost只是实现了而已。
- 目标函数的scalability： 支持不同的loss function, 支持自定义loss function，只要一、二阶可导。有这个特性是因为泰勒二阶展开，得到通用的目标函数形式。

- 学习方法的scalability：Block结构支持并行化，支持 Out-of-core计算（这点和王浩的看法不一样，他写的是优化的trick）

5.4 XGBoost 防止过拟合的方法

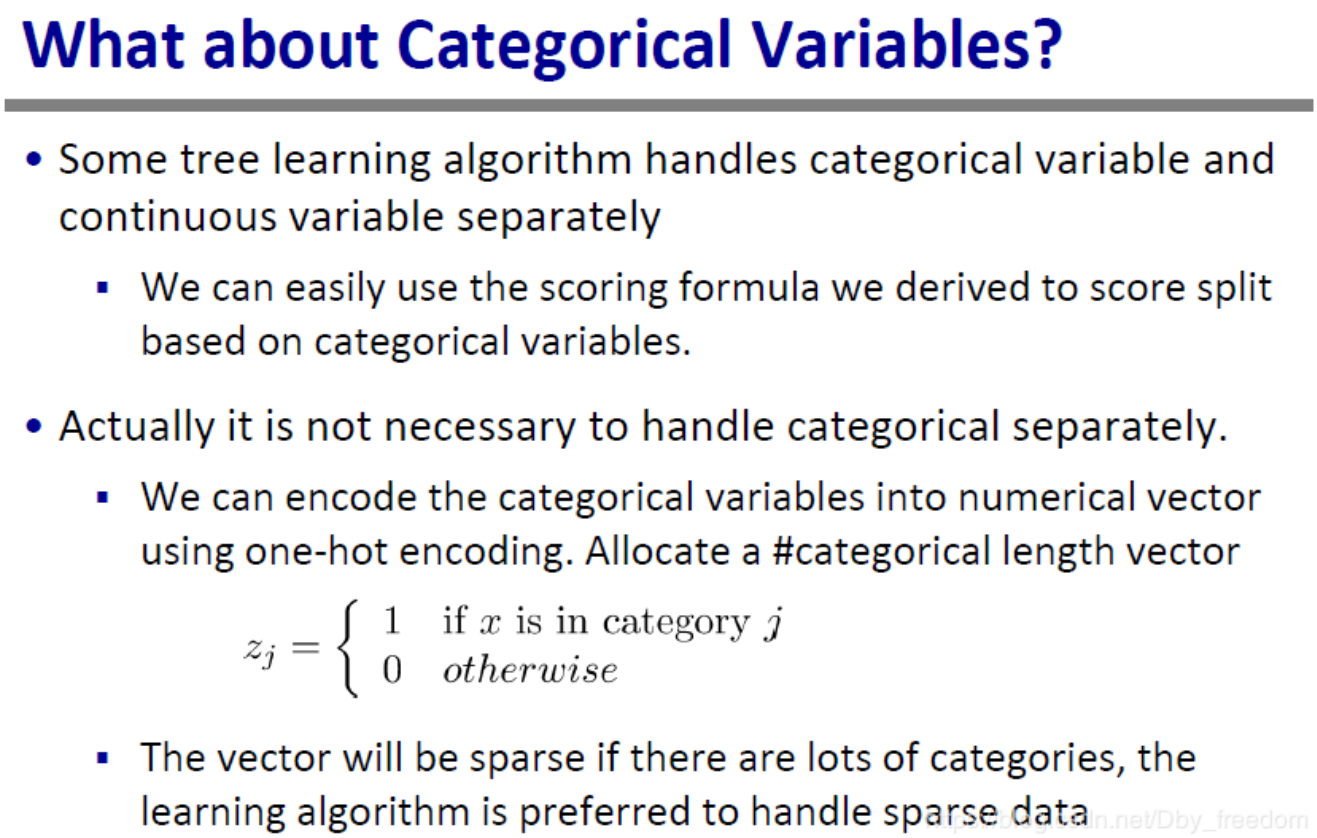
- 目标函数的正则项， 叶子节点数+叶子节点数输出分数的平方和 Ω(ft)=γT+12λ∑Tj=1w2j
- 行抽样和列抽样：训练的时候只用一部分样本和一部分特征
- 可以设置树的最大深度
- η: 可以叫学习率、步长或者shrinkage
- Early stopping：使用的模型不一定是最终的ensemble，可以根据测试集的测试情况，选择使用前若干棵树

5.5 xgboost使用经验总结

- 多类别分类时，类别需要从0开始编码
- Watchlist不会影响模型训练。
- 类别特征必须编码，因为xgboost把特征默认都当成数值型的
- 调参：Notes on Parameter Tuning 以及 Complete Guide to Parameter Tuning in XGBoost (with codes in Python)
- 训练的时候，为了结果可复现，记得设置随机数种子。
- XGBoost的特征重要性是如何得到的？某个特征的重要性（feature score），等于它被选中为树节点分裂特征的次数的和，比如特征A在第一次迭代中（即第一棵树）被选中了1次去分裂树节点，在第二次迭代被选中2次.....那么最终特征A的feature score就是 1+2+....

5.6 类别特征必须编码

无论是XGBoost还是其他的Boosting Tree，使用的Tree都是cart回归树，这也就意味着该类提升树算法只接受数值特征输入，不直接支持类别特征，事实上，对于类别特征的处理，参考XGBoost PPT如下（XGBoost accepts only numeric features）：



而XGBoost具有支持稀疏数据的优势，对one-hot并不抗拒。

> xgboost是不支持category特征的，在训练模型之前，需要我们进行预处理，可以根据特征的具体形式 来选择one-hot encoding（无序）还是label encoding（有序）。 当category的特征值非常多时，one-hot encoding会非常稀疏。这时候one-hot encoding的效果可能不好，可以用NN训练一个该category的向量，或者用其他方式来编码。

5.7 XGBoost并行计算理解

初始学习XGBoost，一个很容易困惑的点就是XGBoost的并行计算，毕竟其采用的加法模型架构，怎么看都是一种串行计算；事实上，XGBoost的并行不是在计算各轮添加的回归树上，而是体现在上述`Split Finding`中的排序上，即并行对各个样本，依据其所有m个特征值进行m轮从小到大排序，然后计算各个特征下各样本从左到右做划分对应的`score`。这才是并行计算的实质所在，由于XGBoost主要的计算量都击中在找`Split Finding`上，故而对这一步的并行计算大大加速了XGBoost的模型运算速度。

关于XGBoost的one-hot编码的一点思考与总结

个人在阅读XGBoost相关文献及问题时候发现关于XGBoost到底是否应该大规模使用one-hot编码存在异议，网上答案也是议论纷纷，个人在这里总结一下结合其他各位答主看法后的自身理解：

> 机器学习中的许多模型中，对类别型变量，常作的处理是，将它们编码成one-hot。但是对于树模型来说，将类别型变量编码成one-hot，这样作是否有意义呢？像一些机器学习工具包（比如：spark gbm实现），你可以指定为类别型变量，内部自己去做one-hot实现。而像xgboost，则将输入全认为是数值型特征去处理。

> 这在机器学习界也有争论。理论上，树模型如果够深，也能将关键的类别型特型切出来。 关于这个，xgboost的作者tqchen在某个[issues]>(https://github.com/dmlc/xgboost/issues/95)有提到过： I do not know what you mean by vector. xgboost treat every input feature as numerical, with support for missing values and sparsity. The decision is at the user So if you want ordered variables, you can transform the variables into numerical levels(say age). Or if you prefer treat it as categorical variable, do one hot encoding. 在另一个issues上也提到过（tqchen commented on 8 May 2015）： One-hot encoding could be helpful when the number of categories are small(in level of 10 to 100). In such case one-hot encoding can discover interesting interactions like (gender=male) AND (job = teacher). While ordering them makes it harder to be discovered(need two split on job). However, indeed there is not a unified way handling categorical features in trees, and usually what tree was really good at was ordered continuous features anyway..

个人总结：

1. 对于类别有序的类别型变量，比如age等，当成数值型变量处理可以的。对于非类别有序的类别型变量，依据类别特征维度选择是做one-hot还是先one-hot再embeding。（one-hot会增加内存开销以及训练时间开销）
2. 类别型变量在范围较小时（tqchen给出的是[10,100]范围内）推荐使用one-hot编码；
3. 对于类别变量范围比较大时，把类别特征转成one-hot coding扔到NN里训练个embedding；比如对于用户的ID，一个大的数据集里面可能有数亿个用户ID，对于这些ID我们可以都映射到一个64维的空间中。模型训练实际上就是更新这个用户ID，在64维的空间中对应的Embedding向量。这样每个用户ID可能包含的信息，都被包含在这个64维的实数向量中了。（比如对于用户的ID，一个大的数据集里面可能有数亿个用户ID，对于这些ID我们可以都映射到一个64维的空间中。模型训练实际上就是更新这个用户ID，在64维的空间中对应的Embedding向量。这样每个用户ID可能包含的信息，都被包含在这个64维的实数向量中了。）

> 此处的embedding：即用nn训练一个categorical feature到label的映射，或直接对类别特征做autoencoder，然后把隐层拿出来直接当特征。 知乎上有一篇对此分析地很好的帖子：https://www.zhihu.com/question/266195966/answer/306104444， 可借鉴参考。

参考文献

[1] Newton Boosting Tree 算法原理：详解 XGBoost

[2] XGBoost论文阅读及其原理

[3] xgboost code insight-1

[4] 『我爱机器学习』集成学习（三）XGBoost

[5] 《XGBoost: A Scalable Tree Boosting System》

[6] 关于sklearn中的决策树是否应该用one-hot编码？

[6][类别型变量onehot编码的那些事](http://d0evi1.com/onehot/)